First-Order Patterns for Information Integration

Mark A. Cameron and Kerry Taylor

CSIRO ICT Centre
GPO Box 664 Canberra, 2601, Australia
{Mark.Cameron,Kerry.Taylor}@csiro.au

Abstract. Advanced inter-enterprise applications operate in an environment that changes rapidly as autonomous services dynamically join and leave a community of interest at will. Well-designed integrated information views simplify inter-enterprise application development by hiding details of data distribution, extraction, filter and transformation from inter-enterprise applications, thereby shielding them from unwanted environment changes. Recently, the local-centric *local-as-view (LAV)* approach to integrated information view specification has attracted attention because of its maintainability advantage over the traditional *global-as-view (GAV)* approach. This paper introduces a first-order predicate calculus mapping language that admits LAV, GAV and *global-and-local-as-view (GLAV)* view mapping specifications over distributed database tables and service functions. Mapping patterns that apply to a wide range of integration problems are presented in the language. A case-study inter-enterprise application is used to illustrate the patterns in action.

1 Introduction

Despite a long and rich research history, data integration remains a key challenge in practice for modern business enterprises. Consumers are demanding that enterprises improve both the efficiency of internal processes and their internal knowledge of the dynamic and historical behaviour of their business. At the same time, rapidly changing business environments require rapid changes in business partner relationships and corporate structures. These two factors put enormous pressure on enterprise information sytems to keep up with the businesses they serve. Inevitably, data integration is required as information is extracted from legacy information systems and recombined and repurposed for new ones.

Lenzerini [1] formalizes an integration system $\mathcal{I}$ as consisting of $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where $\mathcal{G}$ is the global (integrated) schema, $\mathcal{S}$ are source schemas, and $\mathcal{M}$ is a set of assertions relating elements of the global schema with elements of the source schemas. $\mathcal{G}$, $\mathcal{S}$ and $\mathcal{M}$ are specified in some (though possibly different) suitable language. There are four flavours of mapping specification for $\mathcal{M}$: *global-as-view* (GAV), *local-as-view* (LAV), *global-and-local-as-view* (GLAV), and *peer-to-peer*, which we do not address further here.

In a GAV integration system, each mapping in $\mathcal{M}$ defines part of the global schema $\mathcal{G}$ in terms of queries over source schemas $\mathcal{S}$, in much the same way as

D. Lowe and M. Gaedke (Eds.): ICWE 2005, LNCS 3579, pp. 173–184, 2005.
© Springer-Verlag Berlin Heidelberg 2005

one defines a view in SQL as a query over other relations or views. In a LAV setting, each mapping in $\mathcal{M}$ defines a source schema $\mathcal{S}$ in terms of queries over the global schema $\mathcal{G}$. The advantage of LAV mappings is that they treat sources independently: a mapping is phrased in terms of the relationship between a single source schema and the global view, without reference to other sources. GLAV mappings aim to exploit the value of a combined local and global approach. GLAV integration systems admit mappings $\mathcal{M}$ that are a mixture of GAV and LAV mappings, and may relate queries over multiple sources to queries over the global schema.

It is always possible to transform a LAV integration system into a GAV integration system [2], however the reverse is not always possible. The value of a GLAV approach is not well explored [3, 4], but there are two key reasons for preferring it. Despite the advantages of LAV for dynamic source integration, LAV alone does not permit the phrasing of mapping rules that take advantage of within-source joins, nor across-source joins through queries against the global schema.

Whatever style (GAV, LAV, GLAV) of integration mapping is applied, there are specific challenges that integration systems must meet. Data integration, according to [5], involves eight tasks, of which the sixth requires creating mappings between sources and the global schema. This step encapsulates the knowledge gained from all the previous steps, so it is important that the mapping language is sufficiently expressive to deal with the range of possible variation in the understanding developed in the previous steps. It is our goal to ensure that the mappings are both human-readable and directly computer-interpretable in order to actuate the data integration process at run-time (the final step).

In this paper we present such a mapping language, called **iMaPl**, based on the first-order predicate calculus (FOPC). It is a GLAV language, which means that it can express LAV and GAV mappings as degenerate cases. It is a declarative language, which means that it is amenable to interpretation and optimisation by computational means. We present mapping patterns that can be applied in many situations, to assist DBAs or domain experts to formulate the mappings they require for particular data integration projects. We have developed a comprehensive run-time environment for actuation of the mappings for data integration, but this is not presented here.

Outline. The paper is organised as follows. Next we describe a case-study application, Sydney's Information Highway. Then we introduce **iMaPl**. In section 4 we describe problems in information integration and the patterns expressed in **iMaPl** that apply to those problems, illustrated by reference to the case study. We then outline further work and conclude.

2 Case Study: Sydney's Information Highway

For an example in this paper we describe a case study of government information sharing associated with a major transport route in Sydney, Australia, known as

the Sydney Information Highway (SIH) [6]. Local and state government agencies contribute information services to an inter-enterprise application that permits map-based search and display. Basic information services remain under the control and data management responsibility of the originating data custodians, being served to the Web from their own IT infrastructure. Users, including land developers, real estate investors, local government planning authorities and the general public have seamless access to information with a whole-of-government perspective. We introduce a conceptual model of the global schema for Sydney's Information Highway, together with sources and their schemas shown in figure 1.

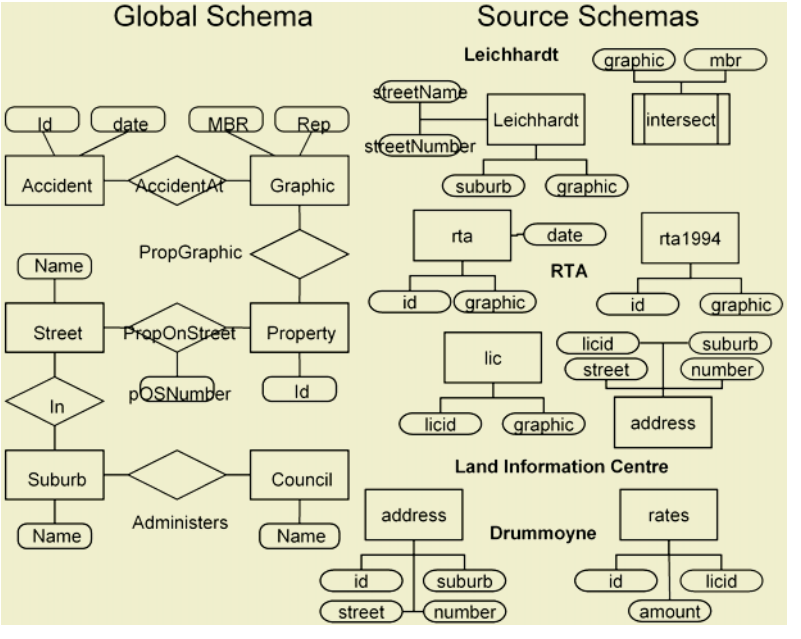

Fig. 1. Schemas

3 iMaPl Mapping Language

3.1 Preliminaries

For this paper, we will favour logic programming terminology and notation as defined in [7]. The **iMaPl** language consists of terms, atomic formulas, and compound formulas. A term is either a constant symbol (we use lowercase symbols, quoted symbols or numbers such as a, "a", 50), a variable symbol (strings starting with uppercase symbols X, or _ for an anonymous variable that is distinct from every other variable), or a compound term consisting of a function symbol and a sequence of terms such as $f(a, X)$. A predicate consists of a predicate symbol (lowercase or quoted symbols such as p or '$p@s$') and a sequence of terms (such as $p(X_1, ..., X_n)$). Constraints are binary operators $Op \in \{<, >, >=, =<\}$ applied to pairs of terms (such as $A < 23$). Compound formulas are formed

in the usual way from predicates, constraints, logical connectives (conjunction $\wedge$, disjunction $\vee$, implication $\rightarrow$ or $\leftarrow$) and quantifiers $Quant \in \{\forall, \exists\}$. Quantified expressions are of the form $\forall(\bar{Var}, P)$ and $\exists(\bar{Var}, P)$ for formula P. By convention, we may omit universal quantifiers. Where notational detail is not important, we simplify term sequences such as $X_1, ..., X_n$ with $\bar{X}$ and indicate omission of detail with '...'.

Describing sources within the $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ model requires a mapping language to connect a language for $\mathcal{S}$ to a language for $\mathcal{G}$. The @ operator ('$p@s$') identifies source predicates from $\mathcal{S}$ distinguishing them from global predicates in $\mathcal{G}$.

3.2 Language Definition

Our mapping language admits three forms of mapping statements: GAV, LAV and GLAV. $\mathcal{M}$ is comprised of assertions of these forms.

Each mapping statement is of the form $Quant(\bar{Var}$,(P $\rightarrow$ Q)). The logical connective implication ($\rightarrow$) partitions the statement into "if [P] then [Q] is a consequence". The implication (Q $\leftarrow$ P) is a notational variation. This provides a convenient way to express what bindings are exchanged between expressions P and Q without having to write expressions using the logical connectives $\sim$, $\wedge$ or $\vee$.

The antecedent [P] and consequent [Q] are compound formulae comprised of predicates, quantifiers, constraints, conjuncts and disjuncts but never including negation or implication. All variables appearing in [Q] must also appear in [P] or else be existentially quantified. Free variables in a mapping statement are assumed to be universally quantified over the scope of the statement.

Using the @ operator to partition our $\mathcal{S}$ and $\mathcal{G}$ schemas, we can now establish syntactic patterns for GAV, LAV and GLAV mapping statements. One of the features of these patterns is that mapping expressions logically constrain models for $\mathcal{G}$ but they do not constrain models for $\mathcal{S}$. That is, mapping statements are unable to create new $\mathcal{S}$ entities–these must come from the sources themselves– and are unable to create inconsistencies in the sources. Therefore they are well suited to autonomous service-based systems. If necessary, integrity constraints can be used to ensure that agreed data quality standards are met.

GAV Mappings. GAV mappings are mapping implications (P $\rightarrow$ Q) where the [P] expression is a compound formula and the [Q] expression consists of only one $\mathcal{G}$ schema predicate. This corresponds to conventional view definitions.

LAV Mappings. LAV mappings are mapping implications (P $\rightarrow$ Q) where the [P] expression consists of one $\mathcal{S}$ predicate; the [Q] expression consists of conjuncts of $\mathcal{G}$ schema predicates. Quantifiers may be used within [P] and [Q].

Applying the LAV mapping scheme to the `leichhardt` schema at the Leichhardt source results in a mapping expression

$$leichhardt(\_, \_, SuburbName, \_)@Leichhardt \rightarrow$$
$$suburb(SuburbName) \wedge suburbName(SuburbName, SuburbName)$$

Which corresponds to the intention "if we obtain a value for `SuburbName` from `leichhardt@Leichhardt` then both `suburb(SuburbName)` and `suburbName(SuburbName, SuburbName)` hold in a model for $\mathcal{G}$".

GLAV Mappings. GLAV mappings are mapping implications (P → Q) where the [P] expression is unconstrained (i.e consists of zero or more $\mathcal{S}$ or $\mathcal{G}$ predicates conjuncts, disjuncts, quantifiers or constraint operators); the [Q] expression consists of conjuncts of $\mathcal{G}$ schema predicates and quantifiers.

4 Integration Patterns

The generic language definition patterns of Sect. 3 provide useful templates for GAV, LAV and GLAV mapping statements. These templates are common building blocks for the problem-centric patterns of schema mismatch, value mismatch and coverage constraints.

Mismatch between source and target schemas and values is a very common problem. Examples of schema mismatch include explicit vs implicit representation of concepts; tabular vs structured representations of entities; and composite vs atomic vs missing entity identifiers. The patterns of Sect. 4.1 are well suited to problems of schema mismatch. Value mismatches cover domain issues such as inconsistent units of measure (m/s vs ft/s), as well as formatting mismatches (capitalized vs lowercase vs camel case) and data entry errors. Section 4.2 documents patterns for addressing value mismatches between sources and the integrated schema. Often both value and schema mismatches occur simultaneously.

4.1 Patterns for Schema Mismatch

Problem: Making Implicit Information Explicit. Successfully incorporating legacy systems into an integrated federation often requires understanding the implicit assumptions developers made. In our example, both Drummoyne and Leichhardt record property information such as street name, number and suburb. Neither system explicitly understands that the suburbs and streets within their domain are administered by the respective council. The `council` entity and the `administers` relationship between Council and Suburb is implicit knowledge for both Leichhardt and Drummoyne.

Pattern: Integration Meta-data. While some data is not available directly from the participating source, we can use our knowledge of the environment to record the data directly within the integration schema. Within a mapping specification $\mathcal{M}$, one might use this GLAV pattern to augment an integrated $\mathcal{G}$ schema predicate with additional database instances that are missing from a source.

Example 1. Integration Meta-data

$$council(leichhardt) \wedge$$
$$councilName(leichhardt, \text{“}Leichhardt\text{”}) \leftarrow$$

Problem: Structural $\mathcal{G}$ Schema Constraints. The token `leichhardt` used to identify Leichhardt as a member of `council` in ex. 1, is somewhat arbitrary, as the service representing Leichhardt cannot itself supply one. However, the relationship between `council` and `councilName` is a $\mathcal{G}$ schema constraint requiring that we use the same token (whatever it may be). Our example dependency is modest, but there are variations of this structural constraint problem that apply to more complex document structures [8, 9].

Pattern: Quantification. Existential $\exists(\bar{Var}, Formula)$ and universal $\forall(\bar{Var}, Formula)$ quantification enables mapping specifications to quantify what is known about individuals that meet criteria expressed in the statement scope. Existential quantification is useful where there are discrepancies in mapping attributes between $\mathcal{S}$ and $\mathcal{G}$ schemas i.e. where the $\mathcal{G}$ schema has more attributes than a source can contribute. More formally, if x is a variable and P is a well formed formula, then $\exists x(P)$ is an existential quantification of x over the scope P; "there exists [x] such that [P]"; and universal quantification "for each [x] such that [P]".

By using the same quantified variable within an expression scope covering structural constraints, it is possible to tie together $\mathcal{G}$ schema entities in a way that satisfies the structural constraints as well as provide values for known information.

Patterns in Action. In example 2, we rewrite the mapping statement of ex. 1 to replace occurrences of the token `leichhardt` with an existentially quantified variable C and simultaneously express a mapping implication to map tokens from our Leichhardt source onto the $\mathcal{G}$ schema entity `suburbName`.

Example 2. Quantification

$$\begin{array}{l}\exists(C,\ (council(C)\wedge\\ councilName(C, \text{“}Leichhardt\text{”})\wedge\\ \forall(SuburbName, leichhardt(\_, \_, SuburbName, \_)@Leichhardt\rightarrow\\ \exists(Suburb, (suburb(Suburb)\wedge\\ suburbName(Suburb, SuburbName)\wedge\\ administers(C, Suburb))))))\end{array}$$

Informally, this mapping expression captures our desire that "there exists a `C` such that `council(C)` and `councilName(C,"Leichhardt")` and if we obtain a value for `SuburbName` from `leichhardt@Leichhardt` then there exists a `Suburb` such that `suburb(Suburb)` and `suburbName(Suburb, SuburbName)` and `administers(C,Suburb)`"

Problem: Transitive Closures. GLAV mappings open the possibility to write recursive mapping statements, for example to generate the transitive closure of a relation. Transitive closures may be needed when dealing with transitive relations obtained from multiple sources [10].

Pattern: Recursive Mappings. Recursive mapping expressions will generally have one or more base (non-recursive) cases, followed by a recursive case. In the example below, we use a recursive mapping pattern to obtain the `connected` relation through functional source operators `nextTo`.

Pattern in Action

Example 3. Recursive Mapping

$$suburb(\text{“}Glebe\text{”})\leftarrow$$

$$suburb(S)\wedge nextTo(S,P)@Leichhardt\rightarrow connected(S,P)\wedge suburb(P)$$

$$suburb(S)\wedge connected(S,P)\wedge nextTo(P,Q)@Leichhardt \rightarrow connected(S,Q)\wedge suburb(Q)$$

4.2 Patterns for Value Mismatch

Problem: Normalized Representation. Our example has shown how to obtain tokens representing suburb names from a source. Leichhardt council provides sentence case names while LIC provides uppercase names. Sometimes a $\mathcal{G}$ schema will require a single representation for tokens for a domain; suburb name in our example has two $\mathcal{S}$ representations, and with a $\mathcal{G}$ schema requirement for lowercase, neither $\mathcal{S}$ representation matches. The challenge for a mapping language is to enable encoding of transformations so that raw data from a source is processed into the normalized representation required by the integrated schema.

Pattern: Source Specific Normalization. The source specific normalization pattern encodes transformation knowledge directly into the $\mathcal{M}$ expression. The idea is that variables from data producers are connected directly to transformation operators, whose output is placed into the $\mathcal{G}$ schema. Source specific normalization follows the scheme: $[P(..., \bar{X})@S_i] \wedge [F(..., \bar{X}, ...\bar{Y})@S_j] \rightarrow [G(...\bar{Y})]$.

Patterns in Action. Example 4 shows this pattern applied to directly transforming Leichhardt source suburb names to lowercase expressed by a within-source join.

Example 4. Source Specific Normalization

$$leichhardt(\_, \_, SuburbNameS, \_)@Leichhardt\wedge$$
$$to\_lowercase(SuburbNameS, SuburbName)@Leichhardt$$
$$\rightarrow$$
$$\exists(Suburb, (suburb(Suburb)\wedge$$
$$suburbName(Suburb, SuburbName)))))$$

This pattern brings up a knowledge-engineering issue. Knowledge about how to manipulate and transform domain values (e.g transform the $\mathcal{S}$ specific domain `SuburbNameS` into the $\mathcal{G}$ schema `SuburbName`) has been tightly merged with knowledge about how to populate a schema from a source.

The following patterns encode an alternative strategy to formulating and writing mapping statements when a separation of knowledge about transforming data representations from populating domains is required.

Pattern: Domain Specific Normalization. In the domain specific normalization pattern, type domains are explicitly described in the $\mathcal{G}$ schema; and we set up domain-specific type transformation schemas, capturing our desire to transform data from one representation into another. These domain specific transformation schemas are global predicates that require implementations. These implementations will of course be delegated to the set of underlying operations offered by sources.

Our source mapping statements will then populate the appropriate $\mathcal{G}$ predicates by appealing to the domain specific transformation offered by the $\mathcal{G}$ schema. This pattern requires recursive mapping expressions. It is similar to techniques described in [10] for dealing with functional dependencies among $\mathcal{G}$ schema predicates; and [11] for dealing with sources with limited capabilities.

In our running example, we have a number of sources offering graphic entities. On closer inspection, one might discover that RTA offers point graphic entities of type `OGISPoint`, Leichhardt offers graphics of type `mifmid`, while LIC offers graphic entities of type `OGISPolygon`. Furthermore, the `functionsRus` service offers a suite of transformation operations: `toMifMid`, `toOgisFeature`, `toSVG` and `toJPG`, each of which is capable of transforming input data from a source type to a target type. The transform services form a directed graph (containing cycles) over the types.

The first part of the pattern requires setting up our domain for graphic entities; we use `typeOf(V, T)` to denote that values of V have type T. This domain functor provides a placeholder for two things: making statements (assertions) about data after it is retrieved from a source or transformed by a service; and acting as a constraint requiring that data be in a specified format. Example 5 shows how this pattern might apply to graphic data elements retrieved from Leichhardt. Similar statement patterns apply for the remaining resources.

Example 5. Populating Domain from Sources

$$leichhardt(..., Graphic)@Leichhardt$$
$$\rightarrow$$
$$graphic(typeOf(Graphic, mifmid))$$

The second part of the pattern requires setting up a `typeTransform` type transformation operator. This operator has two parts: an equality theory for the domain (a statement that says transformations between data in the same type is a no-op and a statement about the transitivity of type transformations); and

statements about the services that implement the operator. Assume a functional dependency from input (first variable) to output. In example 6, the global operator will have implementations delegated to the appropriate service specific operations. Furthermore, notice the use of the domain functor `typeOf(V, T)` as a constraint (or precondition) on the type of information sent to the operation.

Example 6. Domain Rules & Type Transform Implementations

Domain Rules
$typeTransform(typeOf(X,Y), typeOf(X,Y)))\leftarrow$

$typeTransform(X,Y)\wedge typeTransform(Y,Z)$
$\rightarrow$
$typeTransform(X,Z))$

Mapping Rules
$toSVG(Graphic1, Graphic2)@functionsRus\wedge$
$(((Vector = shapeFile)\vee(Vector = mifmid))$
$\rightarrow$
$typeTransform(typeOf(Graphic1, Vector), typeOf(Graphic2, svg)))$
...

Queries over the $\mathcal{G}$ schema use `typeTransform(X,Y)` and `typeOf` to trigger the necessary type conversions. For example:

$$q(Y)\leftarrow graphic(X)\wedge typeTransform(X, typeOf(Y, svg)).$$

Problem: Object Reconciliation. For multiple data sources, shared error-free identifying fields are uncommon. We have shown two techniques, source specific normalization and domain normalization, which are suitable for cases where normalizing is a viable strategy.

Alternatively, value mismatch may be resolved by record linking techniques. There is a large literature focussed on record linkage techniques, software and methods [12–16].

Pattern: Object Reconciliation. In [17], steps for specifying a sequence of virtual services encapsulated in a composition for record linking are given. The virtual service has a $\mathcal{G}$ schema of `link(DataA, DataB, LinkedData)` relying on support predicates such as `standardize`, `index`, `compare` and `classify`. Intuitively, values from two sources are compared and a probabilistic match score is returned indicating the likelihood that values are the same. A pattern for mapping sources to any of the support predicates (F) follows: $[SF_m(..., \bar{X}, ..., \bar{Z})@S_a]\wedge[SF_n(\bar{Z}, ..., \bar{Y})@S_z]\rightarrow[F(step_{i-1}(\bar{X}), step_i(\bar{Y}))]$ The support predicates have function terms such as $step_i$ which work together to ensure that services are sequenced correctly.

4.3 Coverage Constraints

Problem: Logical Fragmentation. A coverage constraint is, informally, a statement about the range of values held for a relation in a particular source database. We should use coverage constraints in query planning to exclude sources from mapped queries when we know from the coverage constraint that those sources cannot contribute to the answers to our query. A coverage constraint is quite different to an integrity constraint, although these can also provide information to assist query planning [18].

Pattern: Coverage Constraint. Constraint expressions are formed from the standard constraint operators $Op \in \{<, >, >=, =<\}$ applied to terms. Our pattern for using coverage constraints requires that the constraints appear on the antecedent of the implication, all variables appearing in the constraint expression are universally quantified (the quantifier may be omitted by convention): $\forall(\bar{X}, [P(..., \bar{X}, ...) \wedge\ Op(\bar{X})] \rightarrow [Q])$

In general, constraint reasoning (see [19] for example) is required to either eliminate mappings from source specific plans or tighten the source specific query bounds.

Patterns in Action. RTA data sources offer accident data based on date. Consider two RTA sources, one contributing data for 1994, and the other for subsequent years. This example demonstrates both coverage constraint and domain specific normalization.

Example 7. Coverage Constraint

$$rta1994(..., Spatial)@rta1994 \wedge$$
$$ADate = 1994$$
$$\rightarrow$$
$$accident(ADate, ..., Spatial) \wedge graphic(typeOf(Spatial, ogisPoint))$$

$$rta(ADate, ..., Spatial)@rta \wedge$$
$$ADate > 1996$$
$$\rightarrow$$
$$accident(ADate, ..., Spatial) \wedge graphic(typeOf(Spatial, ogisFeature))$$

Queries can constrain a predicate thus:

$$q(T) \leftarrow accident(D, ..., S) \wedge typeTransform(S, typeOf(T, svg)) \wedge D > 2000.$$

Constraint reasoning techniques are required to tighten the query bounds (and thus remove the redundant constraint $Date > 1996$ [19]) on the requests sent to sources. The coverage constraint pattern also applies to categorical domains as well as integer domains as we have shown in example 7.

5 Conclusion and Further Work

We have applied recent research on local-as-view query planning to the problem of dynamically building integration plans in response to user-defined requests. Our approach relies heavily on the run-time interpretation of expressive mapping rules that relate sources of data and functional data transformations to global concepts. In this paper we have defined a range of mapping patterns that are designed to address commonly-occurring integration problems. These patterns, expressed in the language of first order predicate calculus, may be interpreted by suitable planning engines, such as one we have under development. Because they are declarative, with a well-defined semantics, they may also be used as a benchmark for evaluating the capabilities of information integration systems.

Our approach subsumes well known GAV, LAV and GLAV mapping patterns for query planning, and extends to offer declarative local database coverage constraints, data type coercion, and recursive mappings. We have defined patterns that address specific information integration problems for schema and value mismatch and object reconciliation.

We are aware of some problems not adequately covered by these patterns, for example, negation, nested implication and mapping expressions that require second-order predicates. Some of these shortcomings in our mapping language arise from our goal for both human readability of the language and sound computational interpretation. We have preferred a syntactic description of the mapping language for ease of adoption by developers at the expense of admitting more expressive mappings that may be safely interpreted only if complex conditions about their construction are met. Ongoing work is extending our pattern set.

References

1. Lenzerini, M.: Data integration: A theoretical perspective. In: PODS, Madison, Wisconsin (2002) 233–246
2. Cali, A., De Giacomo, G., Lenzerini, M.: Models for information integration: turning local-as-view into global-as-view. In: Proceedings of International Workshop on Foundations of Models for Information Integration (10th Workshop in the series Foundations of Models and Languages for Data and Objects). (2001)
3. Lenzerini, M.: Data integration is harder than you thought. Keynote presentation, CoopIS 2001, Trento, Italy (2001)
4. Cali, A., Calvanese, D., De Giacomo, G., Lenzerini, M.: On the expressive power of data integration systems. In: Proc. of the 21st Int. Conf. on Conceptual Modeling (ER 2002). (2002)
5. Seligman, L., Rosenthal, A., Lehner, P., Smith, A.: Data Integration: Where Does the Time Go? Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **25** (2002)
6. Cameron, M.A., Taylor, K.L., Abel, D.J.: The Internet Marketplace Template: An Architecture Template for Inter-enterprise Information Systems. In: Proccedings of the Sixth IFCIS International Conference on Cooperative Information Systems (CoopIS'2001). Volume 2172 of LNCS., Trento, Italy, Springer (2001) 329–343
7. Lloyd, J.W.: Foundations of Logic Programming. 2nd edn. Springer-Verlag (1987)

8. Levy, A.Y., Suciu, D.: Deciding containment for queries with complex objects. In: Proc. of the 16th ACM SIGMOD Symposium on Principles of Database Systems, Tucson, Arizona (1997) 20–31
9. Popa, L., Velegrakis, Y., Miller, R.J., Hernández, M.A., Fagin, R.: Translating web data. In: Proceedings of VLDB 2002, Hong Kong SAR, China (2002) 598–609
10. Duschka, O.M., Genesereth, M.R., Levy, A.Y.: Recursive query plans for data integration. Journal of Logic Programming **43** (2000) 49–73
11. Li, C., Chang, E.Y.: Query planning with limited source capabilities. In: ICDE. (2000) 401–412
12. Fellegi, L., Sunter, A.: A Theory for Record Linkage. Journal of the American Statistical Society **64** (1969) 1183–1210
13. Cohen, W.: Data integration using similarity joins and a word-based information representation language. ACM Transactions on Information Systems **18** (2000) 288–321
14. Elfeky, M., Verykios, V., Elmagarmid, A.: TAILOR: A Record Linkage Toolbox. In: Proc. of the 18th Int. Conf. on Data Engineering, IEEE (2002)
15. Christen, P., Churches, T., Hegland, M.: A parallel open source data linkage system. In: Proc. of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04), Sydney (2004)
16. Nick Koudas, Amit Marathe, D.S.: Flexible string matching against large databases in practice. In: Proceedings of the 30th VLDB Conference, Toronto, Canada (2004)
17. Cameron, M.A., Taylor, K.L., Baxter, R.: Web Service Composition and Record Linking. In: Proceedings of the Workshop on Information Integration on the Web (IIWeb-2004), Toronto, Canada (2004)
18. Cali, A., Calvanese, D., De Giacomo, G., Lenzerini, M.: Accessing data integration systems through conceptual schemas. In: Proceedings of the 20th International Conference on Conceptual Modeling (ER'01), Yokohama, Japan (2001)
19. Guo, S., Sun, W., Weiss, M.A.: Solving satisfiability and implication problems in database systems. ACM Transactions on Database Systems (TODS) **21** (1996)